Parsing in Dialogue Systems using Typed Feature Structures

Rieks op den Akker, Hugo ter Doest,
Mark Moll and Anton Nijholt

Memoranda Informatica 95–25
September 1995

ISSN 0924-3755

University of Twente
Department of Computer Science
P.O. Box 217
7500 AE Enschede
The Netherlands

Order-adress: University of Twente
TO/INF library
The Memoranda Informatica Secretary
P.O. Box 217
7500 AE Enschede
The Netherlands
Tel.: 053-4894021

Parsing in Dialogue Systems using Typed Feature Structures

Memoranda Informatica 95–25

Rieks op den Akker, Hugo ter Doest, Mark Moll and Anton Nijholt
Department of Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
email: {infrieks,terdoest,moll,anijholt}@cs.utwente.nl

September 1995

Abstract

The analysis of natural language in the context of keyboard-driven dialogue systems is the central issue addressed in this paper. A module that corrects typing errors and performs domain-specific morphological analysis has been developed. A parser for typed unification grammars is designed and implemented in C++; for description of the lexicon and the grammer a specialised specification language has been developed. It is argued that typed unification grammars and especially the newly developed specification language are convenient formalisms for describing natural language use in dialogue systems. Research on these issues is carried out in the context of the SCHISMA project, a research project of the Parlevink group in linguistic engineering; participants in SCHISMA are KPN Research and the University of Twente. The aims of the SCHISMA project are twofold: both the accumulation of knowledge in the field of computational linguistics and the development of a natural language interfaced theatre information and booking system is envisaged. The SCHISMA project serves as a testbed for the development of the various language analysis modules necessary for dialogue systems.

1 Introduction

A dialogue system is a system that allows users to communicate with an information system by means of a dialogue using natural language. Such systems always provide their users information about a particular restricted domain, information that is stored in a data or knowledge base, like for instance a travel information system. Within our Parlevink language-engineering project we are currently developing a theatre information and booking system. The sub-project SCHISMA, in which this dialogue system is aimed at, is a joint research project with the Speech and Language group of KPN Research (the R&D department of Royal PTT Nederland). In the projected system the language used to communicate with it is Dutch.

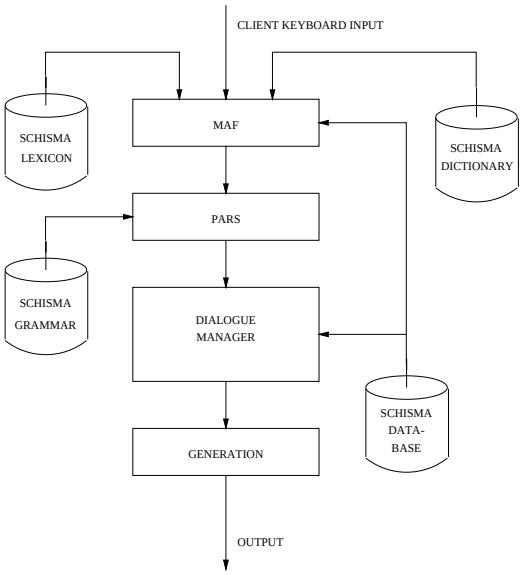


Figure 1: **Global architecture of Schisma**

A dialogue system as meant here has several aspects: extracting the relevant information from the user input, dialogue management, user- and dialogue modelling, information retrieval/update from the database, and language generation. A global architecture of the Schisma dialogue system is shown in figure 1. It is the architecture of a first prototype of the dialogue system. The actual status of a dialogue does not dynamically influence the process of parsing, nor the preprocessing of the input in the module MAF, that handles Morphological Analysis and Fault correction. In this paper we concentrate on the first of the above mentioned aspects implemented in the two modules MAF and PARS. For discussion on dialogue management and user-modelling we refer to (Andernach et al. 1995; Andernach 1995). The system processes user input typed on a keyboard. The MAF module accounts for typing errors in the input, detects word boundaries and use of punctuation and tags the distinguished words with their syntactic type (i.e. category). Details on the MAF module are presented in section 2.

The parser is that part of the dialogue system that should identify the relevant semantic information communicated by the user. It outputs one (ideally) or several (in case of syntactic ambiguities) analyses of the input from the MAF module. The dialogue manager then selects the most likely analysis given the current status of the dialogue. From experiments with a semi-automatic system, in which selected persons set up a dialogue with a Wizard, we gained insight in how client users express their wants, information, and answers in Dutch when they think they are communicating with a machine. Analyses of the corpus of dialogues resulting from these experiments, known as Wizard of Oz experiments, not only give answers to questions like "what words or phrases do clients use" but they also point out that clients make typing errors, sometimes react very unpredictable, and often express themselves by means of ungrammatical but pragmatically well understandable (given the context of the dialogue) sentences. In dialogues in which people ask information about theatre performances in a number of theatres and/or want to book seats for a particular performance, time, date and location phrases very often occur. These domain-dependent characteristics of the SCHISMA dialogues influence the morphological and parse modules that have to concentrate on recognition of these and other domain-specific phrases or words.

Other experiments with existing ("commercially available") natural language interfaces

(Komen 1995) have shown that systems in which input analyses are restricted to the search for particular (domain-dependent) phrases or patterns are often missing relevant information provided by the user. In conclusion, input analysis should do more than this and should try to extract as much syntactic/semantic structure from the input, using general linguistic/grammatical knowledge about the natural language as well as domain specific information concerning the most relevant concepts and their relations. On the other hand we are aware of the fact that a formal syntactical/semantical specification of the user language cannot be complete: users will subject the analyser with input not covered by the formal language, although completely understandable for a human being. Hence, the quest for a robust grammar and parser. In section 3 we will discuss shortly the basic parsing technique and consider robustness.

The quest for a good, i.e. a practically useful, grammar and parsing system is a language engineering exercise consisting of several stages of development out of which, on the basis of experiments with earlier versions, finally and hopefully a satisfying system results. For the development of syntactic/semantic typed unification grammars we have defined and implemented a convenient specification language for typed unification grammars based on a context-free phrase-structure grammar. In sections 4 to 6 our parser and the formal language to specify types, lexicon and grammar is presented. We describe how a syntactic/semantic specification of a fragment of a natural language is compiled and linked with the parser resulting in a parser for the language that translates input sequences into their semantic representation as specifed by the syntactic/semantic specification grammar. In section 4 a general introduction to types and feature structures is given. Unification of typed feature structures is the main and the most costly operation in our parsing technique. For a good performance of the PARS module an efficient implementation of this operation is a prerequisite. Section 5 reports on the way unification is implemented by means of object-oriented techniques. Section 6 presents the specification language for defining feature types, lexicon entries and grammar rules. Finally in section 7 we come to conclusions and present plans for the near future.

The formalisms and techniques presented in this paper are in principal applicable and useful for (the specification of) parsing systems for dialogue systems or natural language interfaces in general. The examples used in this paper come from the SCHISMA application, that functions as the main test environment for gaining insight in the practical value of the efforts and products reported and presented in this paper.

2 The Preprocessor MAF

As we postponed the development of a spoken interface to the SCHISMA system, we concentrate here on the analysis of keyboard input; that is, the input of the MAF module is the character string typed in by the client. The MAF module is best seen as the preprocessor of the SCHISMA system. It handles typing errors and detects certain types of phrases (proper names that occur in the database, date and time phrases, number names, etc.). The latter task of MAF is especially important, since it extracts information crucial for the continuation of the dialogue from the input string.

Output of the MAF module is a *word graph*. We define a word graph here as a directed graph having as its nodes the positions in the input string identified as (possible) word boundaries. Nodes are numbered starting with 0 for the left-most boundary; that is the position left to the first input character. A pair $(index_1, index_2)$ is an edge of the graph if $index_1$ and $index_2$ are word boundaries, $index_1 < index_2$ and the words enclosed between $index_1$ and $index_2$ are identified as one text unit. This implies that edges may provide text units to the parser that, in fact, contain more than one word.

In addition, the MAF module labels the edges of the graph with a value that indicates the quality of the recognition (and, possibly, correction) performed.

```
\begin
/* begin tag */
\orgstring
/* the original string for this item */
\recstring
/* the recognised string */
\type
/* the category/type of the word */
\info
/* additional information to type */
\index1
/* index1 points at a word boundary */
\index2
/* index2 points at a word boundary */
\val
/* probability, costs */
\end
/* end tag */
```

Figure 2: **Output specification of the Maf module;** comments are allowed in between `/*` and `*/`.

2.1 Output of MAF

Of course word graphs are only used on a conceptual level in developing the preprocessor. The word graph is communicated among the distinguished submodules of MAF by means of so called items. The form of these items is as illustrated in figure 2.

On the implementation level this means that the MAF module has as output a collection of items; each item is clearly bounded by `begin` and `end` tokens. Items contain the original string and its reading followed by the `type` (i.e. category) of the string. The `info` token is followed, if required for the feature type at hand, by a canonical representation of the meaning content of the recognised string. For instance, in case of a phrase indicating a date, the `info` field contains a string of format DDMMYY; in case of time phrases the format is HHMM and strings that have type `number` assigned to them have a `info` field of the form DDDD.DD. Recognition of phrases representing database items is

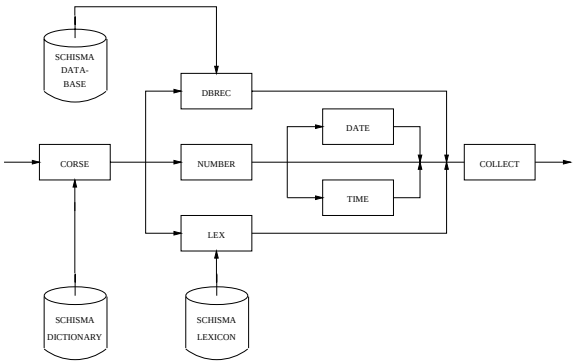


Figure 3: **Architecture of the MAF module**

reflected by a database key occurring after the `info` token. The indices following the `index1` and `index2` tokens specify the boundaries for the string at hand. The value that appears after the `val` token allows us to assign some quality measure to the string recognition. It can be used for choosing the correct analysis of the input string at a later stage (in conjunction with other (preferably) higher level knowledge). The itemset for one word graph is surrounded by the `\beginset` and `\endset` tokens.

The architecture of the MAF module as depicted in figure 3 should now be understood as follows: the error correcting module CORSE (see section 2.2) outputs a word graph that is provided to the tagging modules DBREC, NUMBER, DATE and TIME which scan the graph for phrases that are special in the SCHISMA domain. For details on the tagging modules and the phrases they recognise we refer to section 2.3. In addition, the word graph is provided to the LEX module. For performing the error correction CORSE has access to a large dictionary (typically 200,000 words). The tagging modules look for phrases in the input string that contain particularly important information for the dialogue; especially the detection of proper names referring to database items, phrases indicating date and time information and number names is aimed at here; for detecting proper names referring to the database the DBREC module needs access to the SCHISMA database. LEX searches the word graph for words that appear in the domain-

specific lexicon and determines the appropriate feature type(s) in order to capture their meaning content.

In section 2.4 the LEX module is defined and in section 2.5 some words are devoted to implementation issues.

2.2 Segmentation and Error Correction

Clearly, the analysis of typed input is somewhat simpler than spoken language recognition. However, a typed interface introduces the challenge of handling typing errors the client makes and detecting word boundaries. In detecting and correcting typing errors we have to use knowledge of what character sequences are allowed in Dutch. Roughly there are two approaches to this from the engineering point of view: the *integrated* approach and the *preprocessor* approach. In the integrated approach recognition of tokens (lexical items, number names, etc. (see discussion below)) is done simultaneously with the error correction; this can be done by recording all word components in a trie structure and then operate on it by means of a cost function. See (Oflazer 1994) for details. The preprocessor approach requires the introduction of generic knowledge of what character sequences definitely may and what may not occur. It makes use of what trigrams of characters and what triphones (trigrams of phonemes) are viable in the Dutch language (given a dictionary of words that may occur). Using these trigrams substrings of the input string are compared to words in the dictionary. We refer to (Vosse 1994) for details on this error correction method. For reasons of compositionality we have chosen the latter approach: this option offers the best possibility to partition MAF in a number of submodules that have clear input/output specifications and thus can be developed and implemented separately.

In general, the CORSE module has to account for the following kinds of typing errors:

- insertion of characters,
- deletion of characters,
- substitution of character by other ones,
- exchange of characters; like in `Fikners` (should read `Finkers`).

Clearly, special attention must be given to typing errors concerning word boundaries and the detection of word boundaries themselves. Related to this is that punctuation symbols are handled as one character words.

2.3 Tagging Modules

Clearly, the choice for an error correcting preprocessor is a design decision that has some consequences for the architecture of the MAF component. Most important implication is that the components following the preprocessor, whatever they are have clear input/output relations and may perform their task in sequential order as well as parallel (in contrast to integrated).

The modules described below are in fact specialised taggers; each of them looks for a special type of phrases; if they find the type of phrase they are looking for the phrase is tagged and output to the postprocessor COLLECT. In general the output of the taggers is as illustrated in figure 2. The capitalised literals below correspond to module names in the architecture of MAF as given in figure 3.

- NUMBER; recognition of number names; `40`, `veertig` (fourty), etc.,
- DATE; recognition of date phrases; for instance `morgen` (tomorrow), `vandaag` (today), `12␣februari` (the 12th of February),
- TIME; recognition of time phrases; examples are: `vanavond` (tonight), `om␣acht␣uur` (at eight o'clock),
- DBREC; recognition of the use of proper names occurring in the SCHISMA database; examples: `Youp␣van␣het␣Hek` (proper name of an actor), `Hond␣op␣'t␣IJs` (title of his performance).

The TIME, DATE and DBREC modules are of special importance in the SCHISMA domain, since

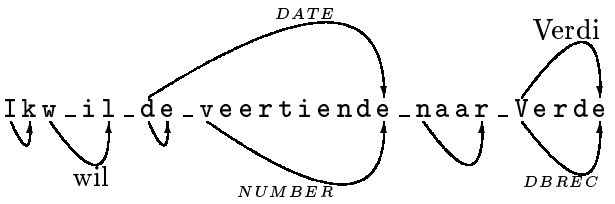


Figure 4: **A word graph of an input string**

they detect phrases on which queries and updates to the SCHISMA database are based. The recognition of number names is considered a task necessary in any dialogue system.

To illustrate how the CORSE and tagging modules work, in figure 4 the word graph of the string `Ikw␣il␣de␣veertiende␣naar␣Verde` (On the 14th I would like to go to Verdi) is given. Spaces are represented by the ␣ character. It can be seen that the CORSE module corrected strings `w␣il` and `Verde` to `wil` (would like) and `Verdi` (the componist), respectively. Also the strings `de␣veertiende` (the 14th), `veertiende` (14th) and `Verde` have been tagged by the DATE, NUMBER and DBREC modules respectively.

2.4 The LEX and COLLECT Modules

The submodule LEX searches the lexicon for words that are provided by the CORSE module through the word graph. The lexicon is a rather small list of words that carry important domain semantics; for each of the words a feature type is supplied in the lexicon. We refer to section 3 for a discussion on typical feature types used for representing meaning in SCHISMA. Before the lexicon actually is searched, a morphological analysis is performed that accounts for inflection of verbs, adjectives, nouns, etc. Strings for which the lexicon does not contain an entry are assigned the (most general) type `word`, a prototype feature type for words not occurring in the lexicon. We refer to section 4 for details on typed feature structures. In addition LEX assigns feature types to punctuation symbols in the word graph (remember that the error correcting module provides these symbols as words in the word graph).

The COLLECT submodule accepts the items as generated by the taggers, the error correcting module and LEX, collects them and surrounds the set by `\beginset` `\endset` tokens to indicate the start and end of the word graph respectively.

2.5 Implementation Report

Currently modules DBREC, CORSE and LEX are available. DBREC and LEX have been developed by the SCHISMA partners, and CORSE has been adapted from the source code based on (Vosse 1994). The other modules DATE, TIME and NUMBER are still under construction.

```
 1  S How may I help you ?
 2  C Are there any tickets left for
 3    the Verdi opera on the 14th
 4    next month?
 5  S Yes, there are,
 6    but only first rank.
 7  C oh that's ok ! What is the Verde
 8    opera about ?
 9  S [description of the plot of
10    the opera]
11  C I'd like two tickets
12  S I will make reservations for two
13    tickets for the Verdi opera on
14    the 14th of October. It's  $40,
15    tickets are $20 each.
16  C ok, thanks
17  S You're welcome, goodbye
```

Figure 5: **Example of a SCHISMA dialogue;** `C` refers to client input and `S` is system output.

In figure 5 an example is given of a SCHISMA dialogue. Referring to the linenumbers in front of the utterances in the dialogue we will show how the distinct submodules and especially the tagging modules act on the client input.

2 `Verdi` is recognised by the DBREC module as a database item; this both restricts the genre (classical music) and the performances the client may aim at (compositions by Verdi); the DATE module detects

the phrase `on the 14th next month` as indicating a date; in figure 6 the item output by MAF for this string is given;

`7,8` the CORSE module corrects the ill-formed string `Verde` to `Verdi`; it therefore uses the SCHISMA lexicon which contains all strings occurring in the database; see figure 6 for the actual output of MAF for this string; the string `opera` is recognised by LEX as a word carrying important domain semantics; in addition the question mark (and other punctuation symbols) are tagged as such by the CORSE module;

`11` the NUMBER module tags the string `two` as a number;

`16` LEX tags the string `ok` and `thanks` with a feature type indicating the end of the dialogue.

```
\begin
\orgstring
Verde
\recstring
Verdi
\type
db  /* db item */
\info
COMP381  /* db key */
\index1
27
\index2
32
\val
.6 /* correction */
\end
```

```
\begin
\orgstring
on the 14th next month
\recstring
on the 14th next month
\type
date
\info
141095  /* DDMMYY */
\index1
47
\index2
69
\val
1.0
\end
```

Figure 6: **Examples of items output by MAF.**

Future research concerning MAF will follow the integrated approach as discussed in section 2.2. Expertise accumulated in working on the separate modules of MAF, will then be incorporated in the new design.

3 Parsing

3.1 Introduction

The input of the parser is the output of the morphological/fault-detecting and -correcting module discussed in the previous section: a word graph, a compact structure representing a set of readings of the (corrected) input from the user, where a reading is a path through the word graph. The parser analyses each of these readings independently of other readings.

Basically the parser is a head-corner chart parser for typed unification grammars. Like all chart parsers it starts with a set of basic items trying to construct completed items covering more and more adjacent words in the input until it has found a complete parse, covering all words. Head-corner parsing can best be seen as a generalisation of left-corner parsing. Both parsing techniques use top-down prediction (unlike pure bottom up chart parsers), but while in left-corner parsing, the input is processed strictly from left to right, in head-corner parsing the parser starts looking for a possible head of the sequence of input words. The set of possible heads of a sequence of words is completely specified by the context-free rules. One of the right-hand side components (nonterminal symbols) of each rule is specified as the head of the rule. It is up to the grammar writer to assign heads to rules. The basic idea is that the head of a rule derives (generates) the most informative (semantically relevant) words of the part of the sentence covered by this rule and that the parser should start looking for these words. An earlier version of this chart parser was presented in (Sikkel and Op den Akker 1993) and we refer to this paper for the technical details about the technique of head-corner chart parsing.

3.2 Robustness

Parsing in the context of a natural language dialogue system should be robust. Robustness here is a property of a feature unification grammar in combination with the parser on the basis of which it works. Robustness means, as far as

natural language processing is concerned, filtering the relevant syntactic/pragmatic information from the reading. We are not interested in a linguistically sound and complete grammmar and parser for Dutch. The syntactic structure of a reading is only of interest as far as it reflects its semantic/pragmatic meaning. Pragmatic meaning is the communicative function of the user utterance in the context of a specific dialogue. Ultimately, it can be expressed in terms of operations (queries and updates) on the system database. Although some parts of a grammar will be useful for dialogue systems for different specific domains and applications, parts of it have to be developed specifically for a particular domain. Hence, a context-free unification grammar is developed on the basis of an analysis of the corpus of dialogues resulting from Wizard of Oz experiments.

For the most important (domain-dependent) phrases and sentence-structures that occur in the corpus, we have developed a context-free unification grammar together with a lexicon of words with typed feature structures. (See the next section for the specification language we use.) Words contained in the lexicon, nouns in particular, have feature types assigned to them that are designed to allow disambiguation by feature unification failure. In developing the grammar we have striven at assigning one analysis to a reading if there is only one meaning. This is quite hard to accomplish especially if the reading contains words that have obtained the default type `word` by the module MAF. This happens if the word could not be matched properly with a word in the domain-specific lexicon or with other domain-specific words. Since the semantic information that goes with these unknown words misses (bottom or $\perp$, see the next section) disambiguation can not be preformed by means of unification failure. The grammar for the SCHISMA prototype we are developing now contains about 100 context-free rules. Each rule is accompanied with a set of feature constraints. These constraints serve two purposes. They specify how to build the semantic feature structure of the left-hand side category compositionally using the feature structures from the right-hand side categories. Also they restrict the applicability of the context-free rule using failure of unification performed on the feature structures of the categories that occur in the rule. Unification is performed during parsing: items on the chart consist of typed feature structures with their indices marking positions in the input string.

The parser outputs for each reading of its input a set of completed items. A completed item is a feature structure and it covers the whole or a part (i.e. a list of subsequent words) of the reading. The part it covers is indicated by the indices of the item. In case of an ambiguous sentence or phrase, it may output several completed items covering the whole reading. It is up to the dialogue manager using the status of the current dialogue to select the most likely parse or partial parse out of the semantic feature structures. The selection of the best parse will also be based on heuristic rules using numbers indicating how informative the covered words are, and on the grammar rules that have been used to build the completed item (analysis).

4 Typed Feature Structures

4.1 Types

Types can be used to categorize linguistic and domain entities. In addition to that the relations between entities can be defined using an inheritance hierarchy. For types we follow the definition of Carpenter (1992). Types can be ordered using the subsumption relation. We write $s \sqsubseteq t$ for two types s and t if s *subsumes* t, that is, s is more general than t. In that case s is called a *supertype* of t, or inversely, t is a *subtype* of s. With the subsumption relation the set of types form a lattice (see figure 7).

The type that subsumes all other types ("the most general type") is called bottom and is denoted by $\perp$. The most general subtype for a pair of types s and t is called the *least upper bound* and is written as $s \sqcup t$. For instance, in figure 7 we have $s \sqcup t = x$ and $v \sqcup w = \top$. In the latter

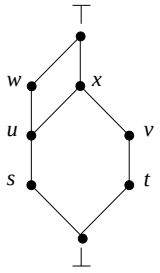


Figure 7: **A type lattice.** The lines represent the subsumption relation. More specific types are placed higher in the lattice. The top element '⊤' is used to denote inconsistency between types.

case the two types contain conflicting information and are hence inconsistent.

There are two ways to specify a type lattice. The first way is to express each new type in terms of its subtypes. This can be seen as a set-theoretical approach: each type is a set of possible values and a new type can be constructed by taking the union of other (possibly infinite) sets. For instance, the type fruit could be defined as

$$\mathsf{fruit} \quad := \quad \mathsf{apples} \cup \mathsf{bananas},$$

where apples is a set of all apples and bananas is a set of all bananas. The bottom element $\bot$ is then the set of all entities within the domain and the top element $\top$ is the empty set $\emptyset$.

The other way to specify a type lattice is to express each type in terms of its supertypes. In this context the term 'inheritance' is often used; a type inherits information from its supertypes. The disadvantage of specifying types this way is that inconsistencies in the lattice are easily introduced. If a type is specified to have two supertypes that contain conflicting information, that type would be inconsistent. With the set-theoretical approach this cannot happen. However, from the grammar writer's point of view it is often easier to first introduce general concepts and later differentiate them into more specific types than to start with the most specific types and generalize over them to construct new types. Hence, in the specification language described in section 6 the second approach is followed.

4.2 Feature Structures

Feature structures provide a convenient way to keep track of complex relations. During parsing constraints can be checked with feature structures, and after parsing the meaning of the language utterance can (hopefully) be extracted from them. The structure of our feature structures is similar to the more traditional form of feature structures as used in the PATR-II system (Shieber 1986) and those defined by Rounds and Kasper (1986).

Typed feature structures are defined as rooted DAGs (directed acyclic graphs), with labeled edges *and nodes*. More formally, we can define a typed feature structure tfs as a 2-tuple $\langle t, features \rangle$, where $t \in Types$, the set of all types, and $features$ is a (possibly empty) set of features. A feature is defined as a feature name / feature value pair. A feature value is again a typed feature structure. At first glance the labels on the nodes seem to be the only difference with the traditional feature structures, but there is more to typing than that. Every type has a fixed set of features. Such a feature value type can be seen as the appropriate value for a particular feature. It should be equal to the greatest lower bound (the most specific supertype) of all the possible values for that feature. So a typed feature structure is actually an instantiation of a type. Types are used as a sort of templates. By typing feature structures we restrict the number of 'allowed' (or appropriate) feature structures. Putting these restrictions on feature structures should fasten the parsing process; at an earlier stage it can be decided if a certain parse should fail.

Another advantage of typing feature structures is that it is no longer necessary to make a distinction between nodes *with* features ('complex nodes'), nodes *without* features ('constant nodes') and nodes with type $\bot$ ('variable nodes') as is often done with traditional feature structures. In a consistent definition of the type lattice the least upperbound of a complex and a constant node should always be $\top$ (unless that constant node represents an abstract, underspec-

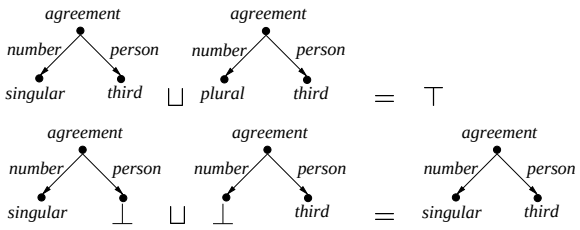


Figure 8: **Basic unifications.** In the first case unification fails, i.e. the feature structures contain conflicting information. The second case is self-evident.

ified piece of information), so that two such nodes can never be unified.

5 Unification

The basic operation on feature structures is unification. New feature structures are created by unifying two existing ones. In figure 8 two basic examples show what unification means. In these feature graphs *agreement*, *singular*, *plural* and *third* are names of types, and *number* and *person* are names of features.

The unification of two feature structures fails if:

- the least upper bound of the two root nodes is $\top$, or
- the unification of the feature values of two features with the same name fails.

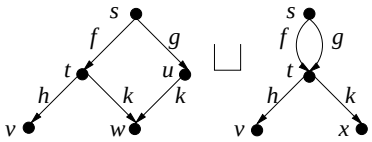
Unification is a costly operation in unification-based parse systems, because it involves a lot of copying of feature structures. In many implementations of parsing systems it takes more than 80% of the total parse time. Several algorithms have been devised to do unification efficiently (Tomabechi 1991; Wroblewski 1987; Sikkel 1993). The efficiency of unification can be increased by minimizing the amount of copying in cases that unification fails, while on the other hand the overhead costs to do this should be as small as possible. Up to now Tomabechi's algorithm seems to be the fastest. With this algorithm the copying of (partial) feature structures is postponed until it has been established that unification can succeed. But Tomabechi already suggests in a footnote that the algorithm can be improved by sharing substructures. This idea has been worked out into an algorithm (Veldhuijzen van Zanten and Op den Akker 1994). The copy algorithm has been implemented in a predecessor of the current parser and has proven to be very effective in experiments. Table 1 shows the results of one of these experiments. The algorithms were tested with the 'sentences' $Jan^n, n = 4 \ldots 8$, using the following grammar: $S \rightarrow S\ S \mid Jan$ (so the sentences are extremely ambiguous). The first column stands for the sentence length, the second column shows the number of parses and the third and fourth column show the number of nodes created during unification for Veldhuijzen van Zanten's and Tomabechi's unification algorithm.

n	parses	HCP	TOM
4	5	100	542
5	14	249	1,827
6	42	662	6,268
7	132	1,897	22,187
8	429	5,799	80,685

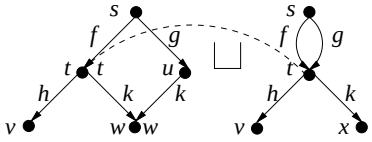
Table 1: **An untyped version of the unification algorithm compared with Tomabechi's algorithm.**

By introducing the types, the overhead increases slightly; for every two nodes that are to be unified the least upper bound has to be looked up in a table. But still, we expect an improvement in the performance.

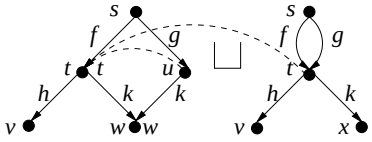
Before the algorithm is described in more detail, it is necessary to define the general properties of a node in a feature structure. These properties ('members' in the object-oriented programming terminology, or 'fields' in a traditional record implementation) can be divided in two kinds: (1) properties that describe the structure of a feature structure and (2) bookkeeping properties, that are used to store intermediate results.



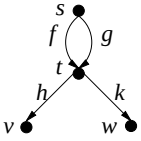
(a) The initial feature structures.



(b) The feature values of f have been unified.



(c) The feature values of g have been unified.



(d) The resulting unifact.

Figure 9: **An example unification.** A type id on the right of a node stands for the auxiliary type of that node. The dashed arrows indicate forward links. If a node has a forward link to another node, the feature structures starting in these nodes are unifiable. It is assumed that $x \sqsubseteq w$ and $u \sqsubseteq t$.

```
proc unify(tfs1, tfs2)
  nextGeneration();
  if unifiable(tfs1, tfs2) then
    return copyUnifact(tfs1)
  else
    return ⊤
  fi
end.

proc unifiable(tfs1, tfs2)
  tfs1 := dereference(tfs1);
  tfs2 := dereference(tfs2);
  if (tfs1 = tfs2) then return true fi;
  tfs1→auxType := lub(tfs1, tfs2);
  if (tfs1→auxType = ⊤) then
    return false
  fi;
  stillUnifies := true;
  while stillUnifies do
        foreach f ∈ tfs2→features
          if (f ∈ tfs1→features) then
            stillUnifies :=
            unifiable(tfs1→f, tfs2→f)
          else
            add feature tfs2→f to
            tfs1→auxFeatures
          fi
        od
  od;
  if (stillUnifies = true) then
    forward(tfs2, tfs1);
    return true
  else
    return false
  fi
end.
```

Figure 10: **The unification algorithm.** An improved version of Tomabechi's quasi-destructive unification algorithm.

For the first kind only

- a type id, that uniquely defines the feature names and the appropiate values for the corresponding features, and
- a set of features, where each feature consists of a name and a value (i.e. an instance of a certain type)

are needed. To handle the bookkeeping we need the following properties:

- a forward pointer: a pointer to another node, of which the unification algorithm has established that unification with this node is possible,
- an auxiliary type id: the type id of the corresponding node in the unifact (the result of unification),
- auxiliary features: features of the node that is unified with the current node, that do not occur in the set of features of the current node,
- an unifact pointer: a pointer to the unifact that is constructed by the copy algorithm,
- a forward mark and an unifact mark: markers containing a generation number indicating whether the forward and unifact pointer can be used in the current unification process.

Unification is executed as a two-step operation: first, it is checked whether unification is possible, that is, the two feature structures to be unified contain no conflicting information. Second, the unifact is constructed using the bookkeeping information left by the first step.

Though the algorithm is implemented with object-oriented techniques in C++, the algorithm is displayed in conventional pseudo-Pascal code to enhance the readability for those not familiar with these techniques. Step one, the check if unification is possible, is shown in figure 10. Some auxiliary procedures for the unification algorithm are displayed in figure 11. Finally, figure 12 shows how step two, the creation of the unifact, is implemented.

The example in figure 9 shows how the unification algorithm works. First the generation counter is increased to make any old intermediate results obsolete. The procedure *unifiable* is then called with the two s nodes as arguments. Now subsequent calls are made to *unifiable* for each feature value pair of the two s nodes. First the feature structures starting in the two t nodes are unified. They differ only in the feature value for the k feature. It is assumed that $x \sqsubseteq w$, so that the two nodes are unifiable. The auxiliary type will then be w. Now the two t nodes are unifiable and a forward link from one t node to the other one can be made (see figure 9b). Now the feature values for the g feature can be unified. Because of the forward link of the previous step, the feature values of the f and g feature of the left feature structure are now unified. So for unification to succeed we have to assume that $u \sqsubseteq t$. Under this assumption a forward link from the u node to the left t node can be made and the initial call to *unifiable* returns **true** (see figure 9c). The final step is then a call to *copyUnifact* to create the unifact from the intermediate results (see figure 9d). Note that this unification is non-destructive; both operands remain intact.

The procedure *lub* (called from *unifiable*) determines the least upper bound of two types. This least upper bound can be looked up in the type lattice as explained in section 4. If two types have $\top$ as least upper bound, they are not unifiable and it is not necessary to look at the feature values of the types.

The procedure *copyUnifact* (figure 12) only creates a new node if it is not possible to share that node with an existing feature structure. A new node is created by *createTFS*, which makes a node of the right type and initializes the features with appropiate values. The variable *needToCopy* is used to check whether a new node has to be created. Only if one of the following two situations occurs it is necessary to make a new node:

```
proc nextGeneration()
  currentGeneration :=
  currentGeneration + 1
end.

proc dereference(tfs_1)
  if tfs→forwardMark = currentGeneration
     ∧ tfs→forward ≠ nil then
    return tfs→forward
  else
    return tfs
  fi
end.

proc forward(tfs_1, tfs_2)
  tfs_1→forward := tfs_2;
  tfs_1→forwardMark := currentGeneration;
end.
```

Figure 11: **Auxiliary procedures for the unification algorithm**

- the unifact has more features than the feature structure from which it constructed, that is, the number of auxiliary features is greater than 0,
- the unifact differs from the feature structure from which it constructed in at least one feature value.

Otherwise the node will be shared with the current node of the typed feature structure from which the unifact is constructed.

6 The Specification Language

To specify a language it is necessary to have a metalanguage to specify that language. Almost always the usage of a specification language is limited to only one grammar formalism. This is not necessarily a drawback, as such a specification language can be better tailored towards the peculiarities of the formalism. For example, ALE (Carpenter and Penn 1994) is a very powerful (type) specification language for the domain of unification-based grammar formalisms.

```
proc copyUnifact(tfs)
  tfs := dereference(tfs);
  if (tfs→unifactMark = currentGeneration)
    then
    return tfs→unifact
  fi;
  needToCopy := (#tfs→auxFeatures > 0);
  i := 0;
  foreach f ∈ tfs→(features ∪ auxFeatures)
    do
    copies[i] := copyUnifact(f);
    needToCopy := needToCopy ∨
    (copies[i] ≠ f);
    i := i + 1
  od
  if needToCopy then
    if tfs→unifact = nil then
      tfs→unifact :=
      createTFS(tfs→auxType)
    fi;
    for j := 0 ... i − 1 do
        add feature copies[j]
        to tfs→unifact
    od;
    tfs→unifactMark := currentGeneration;
    return tfs→unifact
  else
    return tfs
  fi
end.
```

Figure 12: **The copy algorithm.** This procedure generates the unifact after a successful call to *unifiable*(tfs_1, tfs_2).

But apart from expressiveness of the specification language, the ease with which the intended information about a language can be encoded is also important. An example of a language that combines expressiveness with ease of use is the Core Language Engine (Alshawi 1992). Unfortunately the Core Language Engine (CLE) does not support typing. Within our project a type specification language is being developed that can be positioned somewhere between ALE and CLE. This specification language can be used to specify a type lattice, a lexicon and a unification grammar for a head-corner parser. The notation is loosely based on CLE, though far less extensive. For instance, the usage of lambda calculus is not supported.

6.1 Specification of Types

A type specification consists of four parts: a type id for the type to be specified, a list of supertypes, a list of constraints and a formula expressing the semantics for the new type. Figure 13 shows how a type lattice can be specified. For each type `<constraints>` should be replaced with PATR-II-like path equations. Path equations can have the following two forms:

$$\begin{aligned} \langle f_1\ f_2 \ldots f_n \rangle &= \langle g_1\ g_2 \ldots g_n \rangle \\ \langle f_1\ f_2 \ldots f_n \rangle &:= node \end{aligned}$$

The first form says that two paths (i.e., sequences of features) in a feature structure should be joined. With the second form the type of a node in a feature structure can be specified. The right-hand side should be equal to a type identifier or a constant (a string or a number). During the parsing of the specification a minimal satisfying feature structure is constructed for each path equation. So all the nodes in a feature structure have type $\perp$, unless specified otherwise. Subsequent path equations are unified to generate a new feature structure satisfying both constraints. Finally the resulting feature structure for all the constraints is unified with constraints inherited from the supertypes.

`<QLF>` should be replaced with the semantics in a quasi-logical formula. QLF is basically equal to first-order predicate calculus, but is extended with some extra operators to express the mood of an utterance and to express some basic set properties. A more detailed description of QLF can be found in (Moll 1995). The idea is that the constraints are only necessary *during* parsing and the semantics are passed on to be used *after* parsing. In the following example a possible quasi-logical form for a phrase is given:

de opera voorstelling op 4 januari
(the opera performance on 4 January)
`EXISTS X (opera(X) AND date(X,4-1-95))`

The `opera` predicate comes from the QLF part of the opera type and the `date` predicate is generated by parsing the time phrase. Another grammar rule combines these predicates and binds the variable `X`.

A type inherits information from its supertypes in the following way: the constraints for the type are *unified* with the constraints of the supertypes, and the quasi-logical formula for the type is *concatenated* with a list of quasi-logical formulas for the supertypes. The QLF expressions are not evaluated, but are just translated to internal representations.

6.2 Specification of Words

Lexical entries can be specified in the same way as types. This is not surprising, since words can also be seen as types. There is, however, one restriction: a word cannot be used as supertype in the specification of other types (including words). Ambiguous words can be specified by simply defining multiple entries for the same lexeme:

```
LEX("flies", verb, <constraints>, <QLF>)
LEX("flies", noun, <constraints>, <QLF>)
```

The type identifier that is given to a word is the type identifier of the first type of the list of supertypes. In the previous example the words only had one supertype, so there is a feature structure for "flies" with a `verb` type identifier and a feature structure with a `noun` type identifier. In the lexicon every word is associated with a list of feature structures, one for each meaning.

```
TYPE(performance; bottom; <constraints>; <QLF>)
TYPE(play; performance; <constraints>; <QLF>)
TYPE(concert; performance; <constraints>; <QLF>)
TYPE(musical; play, concert; <constraints>; <QLF>)
TYPE(ballet; concert; <constraints>; <QLF>)
```

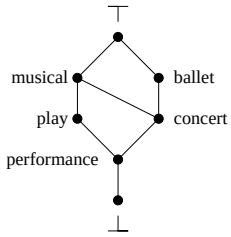


Figure 13: **Specification of a type lattice**

6.3 Specification of Grammar Rules

Grammar rules are internally also represented as typed feature structures. They have a special rule type identifier. It is not necessary to represent rules as feature structures, but such structures happen to be a very practical representation mechanism for grammar rules.

In general, a grammar rule specification looks like the one in figure 14. The asterisks mark the head in the grammar rule. Next to the specification the resulting typed feature structure is shown. Note that grammar symbols are in fact types.

The next example shows how typing can make some grammar rules superfluous.

```
TYPE(perfphrase; nounphrase; ; )
RULE(nounphrase --> *perfphrase*;
  <nounphrase kind> = <perfphrase kind>,
  <nounphrase sem> = <perfphrase sem>;
)
```

The asterisks mark the head in the grammar rule. Both the type and rule specify that a performance phrase is a kind of noun phrase. So with the type specification the rule becomes superfluous.

7 Concluding Remarks

For a practically useful dialogue system full handling of users' natural language input is most important. This is especially the case if the dialogue system does not force the user to provide his/her information in a predefined ("menu driven") order but leaves it to the client to control the dialogue and allows him/her to use free natural language. To find a good formal specification of the language and a robust analyser and parser is a complicated task to be performed before such a dialogue system is obtained. The making of such a system is a process consisting of several design steps. In this paper we have reported on some of the first steps we have taken towards the design of dialogue systems. We consider the development and the implementation of our specification language for typed unification grammars to be a major step in this design. It provides a very convenient tool for the specification of the syntactic and semantic aspects of fragments of natural languages used in dialogue systems. A lot has to be done before the ultimate goal has been reached.

In the near future we will integrate the MAF module in the Wizard of Oz environment, hereby confronting the Wizard with the output of MAF instead of with the user input directly. Experiments with this extended environment will give more insight in the practical value (and shortcomings) of our results. Moreover we will work on an implementation of MAF based on an integrated approach of the users input instead of the "pipe line approach" presented in this paper. Although it will have the same functionality as the one presented here we expect it to be more efficient.

We will also write a second, more complete, version of our grammar and lexicon for the Schisma application using all features provided by the presented specification language for typed unification grammars. The next step is then to incorporate the generated parser into the Wizard of Oz environment as well. The Wizard will then be offered the output of the parsing module: a semantic feature structure that should provide enough relevant information to select an appro-

```
RULE(s --> np *vp*,
     <np agr> = <vp agr>,
     <vp subject> = <np sem>,
     <s sem> = <vp sem> )
```

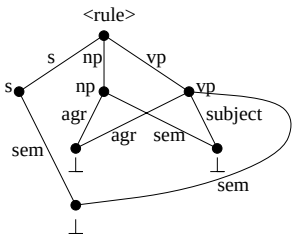


Figure 14: **Specification of a rule**

priate action for continuation of the dialogue. This also offers the opportunity to verify experimentally (using a "real" grammar) our claims about the efficiency of our unification algorithm for typed feature structures. Moreover, these experiments will also be used for the specification of heuristic rules to be used by the dialogue manager in the process of selecting "the best analyses" of user input given the current status of the dialogue.

Acknowledgements

We would like to thank Gert Veldhuijzen van Zanten for making the first major improvements to Tomabechi's unification algorithm.

References

Akker, R. o. d., Ter Doest, H., Moll, M., and Nijholt, A. (1995). Parsing in dialogue systems using typed feature structures. In *Proceedings of the International Workshop on Parsing Technologies*. Prague/Karlovy Vary, Czech Republic.

Alshawi, H., editor (1992). *The Core Language Engine*. Cambridge, MA: The MIT Press.

Andernach, T. (1995). Predicting and interpreting speech acts in a theatre information and booking system. In Andernach, T., Van de Burgt, S. P., and Van der Hoeven, G. F., editors, *Corpus-Based Approaches to Dialogue Modelling, Proceedings of TWLT9*, 107–115. University of Twente. Enschede.

Andernach, T., Ter Doest, H., Op den Akker, R., Schaake, J., Van der Hoeven, G. F., Van der Burgt, S. P., and Nijholt, A. (1995). Language analysis for dialogue management in a theatre information and booking system. In *Language Engineering '95, AI'95, 15th International Conference*, 351–362. Montpellier, France.

Bergin, J. (1994). *Data Abstraction: the Object-Oriented Approach using C++*. New York: McGraw-Hill.

Carpenter, B. (1992). *The Logic of Typed Feature Structures*. Cambridge University Press.

Carpenter, B., and Penn, G. (1994). Ale 2.0 user's guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

Komen, E. (1995). Evaluation of Natural Language™ for the SCHISMA domain. Memoranda Informatica 95-14, University of Twente, Enschede, The Netherlands.

Moll, M. (1995). Head-corner parsing using typed feature structures. Master's thesis, University of Twente, Department of Computer Science.

Oflazer, K. (1994). Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. Technical report, Bilkent University, Ankara, Turkey.

Rounds, W. C., and Kasper, R. T. (1986). A complete logical calculus for record structures representing linguistic information. In

Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, 39–43. Cambridge, MA.

Shieber, S. M. (1986). *An Introduction to Unification-based Approaches to Grammar.* Stanford, CA, USA: Center for the Study of Language and Information.

Sikkel, K. (1993). *Parsing Schemata.* PhD dissertation, University of Twente, Department of Computer Science.

Sikkel, K., and Op den Akker, R. (1993). Predictive head-corner chart parsing. In *Proceedings of the Third International Workshop on Parsing Technologies*, 267–275. Tilburg (The Netherlands), Durbuy (Belgium).

Stroustrup, B. (1991). *The C++ Programming Language.* Reading, MA: Addison-Wesley. second edition.

Tomabechi, H. (1991). Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the ACL.* Berkeley, CA.

Veldhuijzen van Zanten, G., and Op den Akker, R. (1994). Developing natural language interfaces: a test case. In Boves, L., and Nijholt, A., editors, *Twente Workshop on Language Technology 8: Speech and Language Engineering*, 121–135. Enschede, The Netherlands.

Vosse, T. G. (1994). *The Word Connection.* PhD dissertation, Rijksuniversiteit Leiden. Neslia Paniculata.

Wroblewski, D. (1987). Nondestructive graph unification. In *Proceedings of the Sixth National Conference on Artificial Intelligence.*